Перегрузка операторов

Си++ позволяет вам легко вводить новые типы данных. Так, например, вы можете определить класс для работы с комплексными числами или числами в полярной системе координат. Естественно, что удобнее всего проводить вычисления с объектами таких классов при помощи операторов, а не специальных методов или функций.

В Си++ вы можете переопределить большинство операторов языка для работы с вашими типами данных. Вот список операторов, которые вы можете переопределить:

! = < > += –=
!= , –> –>* & |
( ) [ ] new delete >> <<=
^= &= |= << >>= ==
~ *= /= %= % ^
+ - * / ++ --
<= >= && ||

Переопределение операторов вызывает перегрузку операторов. Как в случае перегруженных функций и методов, перегруженные операторы вызываются в зависимости от количества и типа их параметров. О перегруженных функциях вы можете прочитать в разделе “Перегрузка имен функций”.

Для переопределения операторов предназначено ключевое слово operator. Вы должны определить функцию или метод, имя которого состоит из ключевого слова operator и самого оператора. Параметры этой функции должны соответствовать параметрам переопределяемого оператора.

Вы можете переопределить оператор для объектов класса, используя соответствующий метод класса или дружественную функцию класса. Когда вы переопределяете оператор с помощью метода класса, то в качестве неявного параметра метод принимает ключевое слово this, являющееся указателем на данный объект класса. *Поэтому если переопределяется бинарный оператор, то переопределяющий его метод класса должен принимать только один параметр, а если переопределяется унарный оператор – метод класса вообще не должен иметь параметров.*

Если оператор переопределяется при помощи дружественной функции, то он должен принимать два параметра для бинарных операторов и один параметр для унарных операторов.

Существует ряд ограничений, которые необходимо учитывать при переопределении операторов:

• Нельзя изменять количество параметров оператора. Например, нельзя переопределить унарную операцию как бинарную и наоборот

• Нельзя изменять старшинство операторов

• Нельзя определять новые операторы

• Нельзя переопределять операторы принимающие в качестве параметров стандартные типы данных языка, такие как int или char

• Переопределенные операторы не могут иметь параметров, используемых по умолчанию

• Нельзя переопределять следующие операторы: (.), (.*), (::), (?:), а также символы, обрабатываемые препроцессором (символы комментария и т. д.).

Унарные и бинарные операторы

Различают три основных вида операторов: унарные, бинарные и *n*-арные (*n*>2).
Унарные операторы – это операторы, которые для вычислений требуют одного операнда, который может размещаться справа или слева от самого оператора.
Примеры унарных операторов:
i++
--a
-8

Бинарные операторы – это операторы, которые для вычисления требуют двух операндов.

Пример. Ниже отображены фрагменты выражений с бинарными операторами **+**, **–**, **%**, ***** *n*-арные операторы для вычислений требуют более двух операндов.

a+b
f1-f2
c%d
x1*x2

В языке C++ есть тернарная операция **?:**, которая для своей работы требует три операнда

Перегрузка оператора ссылки на член объекта ->.Операторная функция operator->()

В C++ оператор доступа к члену объекта -> можно перегружать. Если оператор -> перегружен, то вызов элемента класса имеет следующий общий вид

obj->item

где

- obj – объект класса;
- item – некоторый элемент класса (внутренняя переменная, метод). Этот элемент должен быть членом класса, который доступен внутри объекта.
- При перегрузке следует учитывать следующие особенности:
- оператор -> считается унарным;
- операторная функция operator->() должна возвращать указатель на объект класса, для которого он определен;
- операторная функция operator->() должна быть членом класса, для которого она реализуется.

Общая форма класса, в котором перегружен оператор ->, следующая

```
class ClassName
{

   // ...

   // операторная функция
   ClassName* operator->()
   {
      // тело операторной функции
      // ...

      return this; // возвратить указатель на объект класса
   }
};
```

Перегрузка оператора -> в классе, содержащего одиночную внутреннюю переменную типа double

Объявляется класс Double, содержащий внутреннюю переменную d типа double. В классе реализована перегруженная функция operator->(), перегружающая оператор ->.

```
#include <iostream>
using namespace std;
class Double
{
public:
```

```
    double d; // внутренняя переменная

    // операторная функция, перегружающая оператор ->
    Double* operator->()
    {
        return this; // возвратить указатель на класс
    }
};

void main()
{
    // перегрузка оператора доступа по указателю ->
    Double D; // экземпляр класса D
    double x;

    D.d = 3.85;

    // вызов операторной функции operator->()
    x = D->d;

    // другой способ доступа
    x = D.d; // x = 3.85

    cout << "x = " << x;
}
```

Результат выполнения программы

```
x = 3.85
```

Перегрузки оператора -> для класса, содержащего динамический массив экземпляров класса

В примере демонстрируется перегрузка оператора -> для класса, в котором реализован динамический массив объектов (экземпляров) класса.

Задан класс Point, реализующий точку на плоскости. В классе Point реализованы следующие элементы:

- внутренние переменные x, y – координаты точки;
- конструкторы;
- методы доступа GetX(), GetY(), SetXY().
- Также задан класс Polygon, который реализует массив точек. В классе объявляются:
- внутренняя скрытая (private) переменная-указатель на тип Point. Эта переменная содержит указатель на динамический массив точек типа Point;
- внутренняя скрытая (private) переменная n, определяющая количество элементов массива;
- конструктор;
- метод GetN(), что предназначенный для получения значения n;
- метод Add(), который добавляет новую точку типа Point к массиву. Новая точка получается как параметр метода;
- метод Delete(), который удаляет точку с заданной позиции index. Позиция index есть входным параметром метода;
- метод GetPoint(), возвращающий значение точки типа Point в заданной позиции index;

- метод Show(), который отображает на экране значение массива точек в классе Polygon;
- операторная функция operator->(), которая перегружает оператор ->.

Текст программы типа Console Application, содержащий реализации классов Point и Polygon, следующий

```
#include <iostream>
using namespace std;

// класс, который описывает точку
class Point
{
private:
   double x, y; // внутренние переменные

public:
   // конструкторы класса
   Point()
   {
      x = y = 0;
   }

   Point(double x, double y)
   {
      this->x = x;
      this->y = y;
   }

   // методы доступа
   void SetXY(double x, double y)
   {
      this->x = x;
      this->y = y;
   }

   double GetX(void) { return x; }
   double GetY(void) { return y; }
};

// класс многоугольник
class Polygon
{
private:
   Point * Pt; // массив точек
   int n; // количество точек

public:
   // конструктор
   Polygon()
   {
      n = 0;
      Pt = NULL;
   }
```

```
// методы доступа
// считать количество точек
int Get(void) { return n; }

// добавить новую точку
void Add(Point p)
{
   Point * Pt2;

   // выделить память под новый массив типа Point - на 1 элемент больше
   Pt2 = new Point[n + 1];

   // скопировать Pt=>Pt2
   for (int i = 0; i < n;i++)
      Pt2[i].SetXY(Pt[i].GetX(), Pt[i].GetY());

   // добавить лишний элемент
   Pt2[n].SetXY(p.GetX(), p.GetY());
   // освободить память, выделенную под старый массив
   if (n > 0)
      delete[] Pt;
   // присвоить внутренней переменной новый массив
   Pt = Pt2;
   n++;
}
// удалить точку в позиции pos
void Delete(int pos)
{
   if (n < 0) return;
   if (pos > (n - 1)) return;
   if (pos < 0) return;
   // если один элемент, то удалить его
   if (n == 1)
   {
      n = 0;
      delete[] Pt;
      return;
   }
   // n>1
   Point* Pt2; // новый массив
   double tx, ty; // дополнительные переменные
   Pt2 = new Point[n - 1]; // выделить память под новый массив - на 1 элемент меньше
   // скопировать Pt в Pt2 в обход позиции pos
   // до позиции pos
   for (int i = 0; i < pos; i++)
   {
      tx = Pt[i].GetX();
      ty = Pt[i].GetY();
      Pt2[i].SetXY(tx, ty);
   }
   // после позиции pos
```

```
        for (int i = pos + 1; i < n; i++)
        {
            tx = Pt[i].GetX();
            ty = Pt[i].GetY();
            Pt2[i-1].SetXY(tx, ty); // Pt2[i-1] - важно
        }
        // освободить память, выделенную для Pt раньше
        delete[] Pt;
        // уменьшить количество элементов на 1
        n--;
        // установить новое значение Pt
        // перенаправить Pt на Pt2
        Pt = Pt2;
    }
    // взять точку с позиции index
    Point GetPoint(int index)
    {
        if ((index >= 0) && (index < n))
            return Point(Pt[index].GetX(), Pt[index].GetY());
        else
            return Point(0.0, 0.0);
    }
    // метод, который выводит массив
    void Show(void)
    {
        double x, y;
        int i;
        for (i = 0; i < n; i++)
        {
            x = Pt[i].GetX();
            y = Pt[i].GetY();
            cout << "x" << i + 1 << " = " << x << ", ";
            cout << "y" << i + 1 << " = " << y;
            cout << endl;
        }
    }
    // операторная функция, которая перегружает оператор ->
    Polygon* operator->()
    {
        return this;
    }
};

void main()
{
    // перегрузка оператора доступа по указателю ->
    Polygon Pol; // экземпляр класса Polygon
    double x, y;
    int i;
    Point Pt;
    // сформировать произвольный массив из 5 точек
    for (i = 0; i < 5; i++)
```

```
   {
      x = (double)(i * 2);
      y = (double)(i + 3);
      Pt.SetXY(x, y); // сформировать точку
      // вызов операторной функции operator->
      Pol->Add(Pt); // добавить точку
   }
   cout << "Array of points: \n";
   // вывести массив через обращение к операторной функции operator->()
   Pol->Show();
   // удалить точку в позиции 2
   Pol->Delete(2);
   cout << "Modified array: \n";
   cout << "n = " << Pol->Get() << endl;
   // снова вывести массив
   Pol->Show();
}
```

Результат выполнения программы

```
Array of points:
x1 = 0, y1 = 3
x2 = 2, y2 = 4
x3 = 4, y3 = 5
x4 = 6, y4 = 6
x5 = 8, y5 = 7
Modified array:
n =4
x1 = 0, y1 = 3
x2 = 2, y2 = 4
x3 = 6, y3 = 6
x4 = 8, y4 = 7
```

Перегрузка оператора ‘ , ‘ (запятая

В языке C++ оператор ‘ , ‘ может быть перегружен. При перегрузке оператора ‘ , ‘ в классе должна быть объявлена операторная функция operator,(). В тело операторной функции можно поместить любой код. При желании оператор ‘ , ‘ может выполнять любые нестандартные операции над объектами класса.

В стандартном случае оператор ‘ , ‘ используется в операции присваивания по образцу

obj1 = (obj2, obj3, ..., objN);

где obj1, …, objN – экземпляры некоторого класса.

В случае стандартного использования оператора ‘,’ нужно учесть следующие особенности:

- оператор ‘ , ‘ считается бинарным. Поэтому операторная функция operator,() получает один параметр;
- при использовании перегруженного оператора ‘ , ‘ в операции присваивания принимается ко вниманию последний аргумент (этот аргумент есть результатом операции). Все другие аргументы игнорируются.

В *общем* случае при перегрузке оператора ‘ , ‘ класс имеет следующий вид

```
class ClassName
{

   // ...

   // операторная функция, которая перегружает оператор ','
```

```
    ClassName operator,(ClassName obj)
    {
        // ...
    }
};
```

где

- *ClassName* – имя класса, в котором перегружается оператор **‘ , ‘**;
- *obj* – имя экземпляра класса, который передается как параметр в операторную функцию operator,().

Перегрузка оператора ‘ , ‘

В примере перегружается оператор ‘,’ в классе Coords3D, который реализует координаты в пространстве. В классе объявляются:

- три внутренние скрытые (private) переменные с именами x, y, z;
- два конструктора класса;
- метод доступа Get(), предназначенный для получения значений x, y, z;
- операторная функция operator,(), которая перегружает оператор ‘ , ‘.

Вид программы для приложения типа Console Application следующий

```
#include <iostream>
using namespace std;
// класс, который определяет координаты точки в пространстве
class Coords3D
{
private:
    double x, y, z;

public:
    Coords3D()
    {
        x = y = z = 0;
    }

    Coords3D(double x, double y, double z)
    {
        this->x = x;
        this->y = y;
        this->z = z;
    }

    // метод чтения x, y, z
    void Get(double& x, double& y, double& z)
    {
        x = this->x;
        y = this->y;
        z = this->z;
    }

    // перегруженный оператор ,
    Coords3D operator,(Coords3D obj)
    {
        Coords3D tmp;
        tmp.x = obj.x;
        tmp.y = obj.y;
```

```
        tmp.z = obj.z;
        return tmp;
    }
};

void main()
{
    double x, y, z;
    Coords3D c1(1, 3, 5); // экземпляры класса Coords3D
    Coords3D c2(2, 4, 6);
    Coords3D c3;

    // вызов операторной функции c3.operator,(c2)
    c3 = (c1, c2); // в c3 записывается c2

    // проверка
    c3.Get(x, y, z); // x = 2, y = 4, z = 6

    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    cout << "z = " << z << endl;

    //------------------------
    //создать другой экземпляр
    Coords3D c4(10, 15, 20);

    c3 = (c2, c1, c4); // c3 <= c4

    // проверка
    c3.Get(x, y, z); // x = 10, y = 15, z = 20

    cout << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    cout << "z = " << z << endl;
}
```

В вышеприведенном коде в строке

c3 = (c1, c2);

вызывается операторная функция c3.operator,(c2). Значит, к вниманию принимается последний экземпляр c2. Экземпляр с именем c1 игнорируется. Это касается и строки

c3 = (c2, c1, c4);

где объекту c3 присваиваются значение внутренних переменных объекта c4.

Вывод: в случае стандартного использования, при перегрузке оператора ‘ , ‘ к вниманию принимается последний справа аргумент. Все другие аргументы игнорируются. Однако, следует учесть, что каждое выражение в последовательности разделенной ‘ , ‘ вычисляется компилятором что необходимо тоже учитывать.

Результат работы программы

```
x = 2
y = 4
z = 6

x = 10
```

y = 15
z = 20